Application Note

The Z380—A 16, 32-Bit Z80® CPU

AN008602-0708

INTRODUCTION

Zilog's Z80® 8-bit CPU, with its powerful instruction set, fast execution time and extensively installed software base, continues to be extremely popular in today's 32-bit world. The Z80's widespread use includes applications in toys, printers, faxes, modem, data communications equipment, and wireless technology.

In today's marketplace, products undergo continuous redesign during their life cycle. New features are often added in response to competitive pressures. Although it is impossible to design a product that can be expanded indefinitely, the design should be open-ended to prevent having to start over from scratch. Software development is always the most expensive, and hence the most valuable, high-risk element of the project's design. For this reason the engineer must ensure that the processor is not operating at peak capacity. Being forced to change a CPU in midstream development due to software bottlenecks will drive costs up dramatically. Additionally, in real-time embedded applications, where much of the software is done in assembly language (processor dependent code), the cost of making a change will be even more significant.

Only three major processor families give the designer a wide range of performance with upgrade-compatible instruction: Motorola's 68000/20/30/40 CPU32 series spans the range of performance to 32 bits; Intel's 8068/286/386/ 486/Pentium, and I960 family. The third major processor family, Zilog's Z80/Z180/Z380, gives the user a choice of compatible CPUs with 8-, 16-bit bus versions and a wide selection of peripherals. However, apart from the selection of processor, other issues are important when considering the total cost of the design. These issues include peripheral availability, time to market, lower power consumption and development costs.

Zilog's new, high-performance Z380 processor is designed for today's sophisticated embedded-control applications. In addition to providing a natural upgrade path for Z80/Z180 applications, the Z380's unique architecture makes it ideal for multitasking embedded-control applications.

The Z380 CPU incorporates advanced architectural features that allow fast and efficient throughput and increased memory addressing capability while maintaining Z80®/ Z180® object code compatibility.

The Z380 CPU is an enhanced version of the Z80 CPU. The Z80 instruction set has been retained, adding a full compliment of 16-bit arithmetic and logical operations, multiply and divide, a complete set of register-to-register loads and exchanges, plus 32-bit load and exchange, and 32-bit arithmetic operations for address calculations.

The addressing modes of the Z80 have been enhanced with Stack pointer relative loads and stores, 16-bit and 24-bit indexed offsets, and more flexible indirect register addressing. All of the addressing modes allow access to the entire 32-bit addressing space.

The register set of the Z80 microprocessor is expanded to 32 bits, and has been replicated four times to allow for fast context switching among tasks in a dedicated control environment.

Key features of Z380 include:

- Full static CMOS design with low-power standby mode support
- 32-bit internal data paths and ALU
- 16-bit (64K) or 32-bit (4G) linear addressing space
- 16-bit internal data bus
- Two clock cycle minimum instruction execution
- Two clock cycle Memory bus
- Programmable I/O bus protocols and clock rates
- Four banks of 32-bit registers
- Enhanced interrupt capabilities, including 16-bit vectors and four external Interrupt inputs
- Undefined opcode trap for full Z380 CPU instruction set

(Continued)

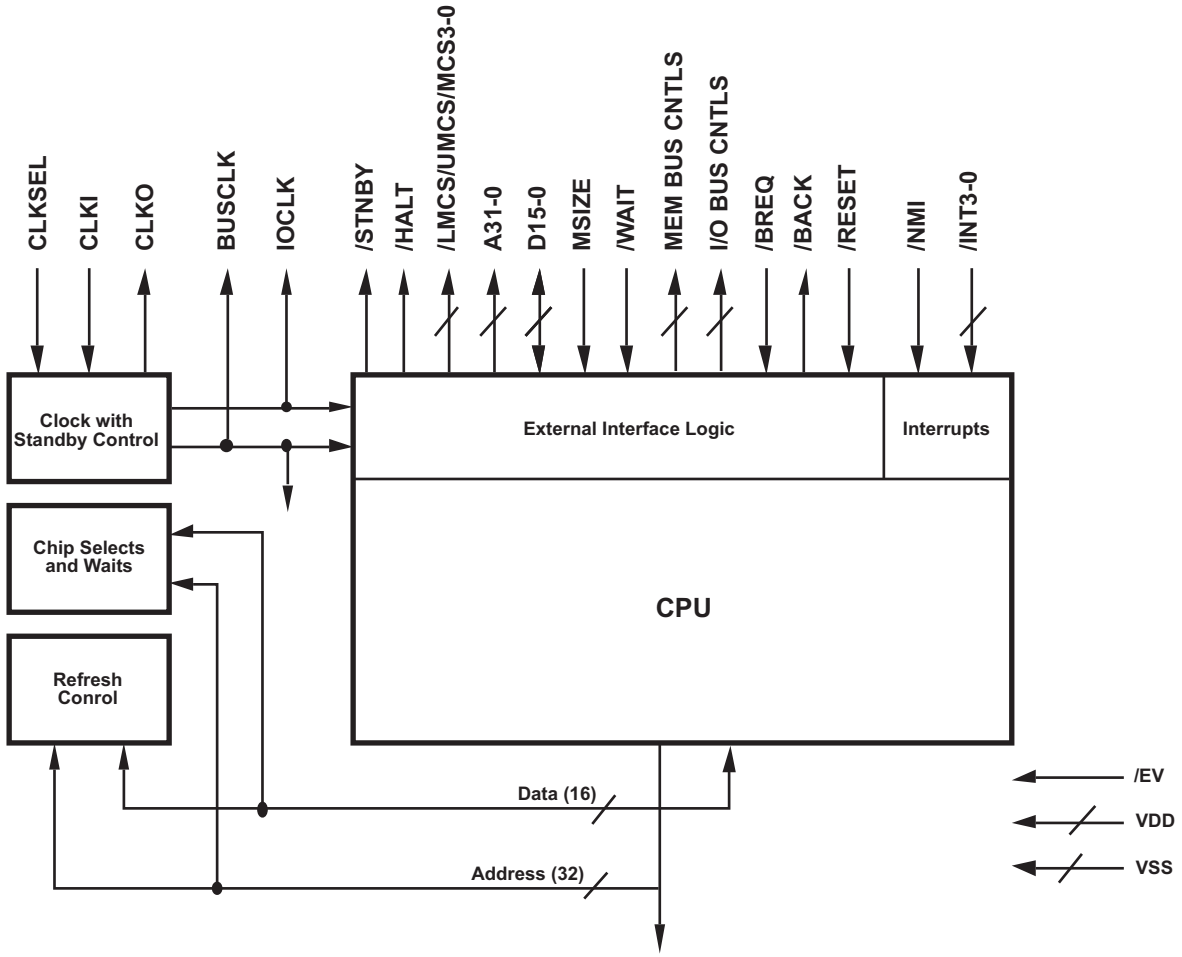

Figure 1. Z380 Functional Block Diagram

BENCHMARKS AMONG EMBEDDED PROCESSORS

In response to a recent microprocessor selection process by a major customer, Zilog's Datacom Marketing group compared the performance and program memory requirements among the new Z80380 and several competing processors, including the Intel 80186 and 80960 and the Motorola 68020 and CPU32. (The CPU32 is the heart of the Motorola's 683xx series of integrated products.)

METHOD

Benchmarking consisted of selecting four code fragments judged to be typical of embedded applications, coding the four fragments in assembly language for each of the four processors, and calculating the execution time for each fragment on each processor, at 16, 25, and 40 MHz clock rates as applicable to each.

The results were then tabulated in a spreadsheet that first normalized them to the figure for the 25 MHz 80380, and then averaged the normalized values for memory code size and execution time, as well as an overall "figure of merit".

The code fragments were called "I/O Loop", "Signed Byte Handling". "Multiply/Accumulate". and "Interrupt". Since the execution time for I/O Loop is a function of the number of times through the loop, and because it was felt to be the most typical of user requirements, it was counted twice toward the composite performance and merit figures, once for a single iteration and once for eight times through the loop. Finally, a fifth performance category was included, the time required for memory-to-memory block movement of data. This made six performance values that were averaged with four program-size values for the overall Figure of Merit, an imbalance that "felt right" in terms of the way we think many users view the value of an embedded microprocessor.

ASSUMPTIONS

Because execution time can be a complex matter for today's pipelined processors, our benchmarks made several assumptions that simplified performance evaluation. The most presumptive was that the memory on all processors was fast enough that there would be NO WAIT STATES. (In many cases this would mandate fast Static RAM rather than larger, more economical Dynamic RAM, which makes sense for some applications but not others.)

A second assumption was that all operands were ALIGNED to the natural boundaries for their size: data accessed 16 bits at a time was located at an address that was a multiple of two bytes, while data to be accessed 32 bits at a time was located at an address that's a multiple of 4 bytes. This characteristic can be guaranteed by many high-level-language compilers, and is questionable only for the Block Move operations.

For processors that include a cache (the 68020 and 80960), the timing was calculated such that the first access to each instruction was a cache miss, and any subsequent accesses were cache hits. In other words, we assumed that these code fragments were not part of a central loop, but were executed in response to specific events that were sufficiently infrequent that the code was superseded in the cache between events.

INSTRUCTION TIMING

For the 80186, we allowed Intel their stated timing assumption "With a 16-Bit Bus Interface Unit, the 80186 has sufficient bus performance to ensure that an adequate number of pre-fetched bytes will reside in the queue most of the time." (16-32 Bit Embedded Processors 1990, pp. 1-50, 1-118). The following 80186 listings include a column of the number of clocks for each instruction, taken directly from the referenced data book.

Motorola's CPU32 User Manual includes several figures for each instruction and addressing mode, which have to be combined with each other and

with those for the following instruction, to determine execution times. The symbols Cea, Hea, and Tea represent the total number of clocks to fetch or calculate an Effective Address, and how many of these represent Header clocks that can be overlapped with subsequent operations, and Tail clocks that can be overlapped with subsequent operations. Similarly, the symbols Cop, Hop, and Top represent the total number of clocks needed to execute the instruction, and how many of these are Header clocks and Tail clocks. The total was computed by the formula:

Cea - min (Tea, Hop) + Cop - min (Top, Hea [next instruction])

For instructions containing two effective addresses the formula is: Cea1 - min (Tea1, Hea2) + Cea2 - min (Tea2, Hop) + Cop - min (Top, Hea [next instruction])

Each following 680x0 code fragment is followed by a spreadsheet that performs these calculations for the CPU32.

For the 68020, Motorola gives three timing figures for each instruction. Best Case (BC) is the number of clocks the instruction takes if it is in the cache and enjoys the maximum possible degree of overlap with the preceding and following instructions. Cache case (CC) is the number of clocks is the instruction is in the cache but has no overlap with other instructions. Worst case (WC) is when the instruction is not in the cache and has no overlap with other instructions.

The 68020 User Manual includes quite a few pages that define these three timings for all the possible instruction variants, but then notes that there is no way to use these values to arrive at actual execution times! Since CC-BC is the maximum possible instruction overlap, we decided to count the first execution of an instruction as a cache miss with an "average" amount of overlap:

first execution = WC - (CC-BC)/2

while subsequent executions of an instructions were counted as a cache hit with an average amount of overlap:

subsequent execution = (CC+BC)/2

Each following 680x0 code fragment is followed by a spreadsheet that performs these calculations for the 68020.

For the 80960, an actual clock-by-clock analysis of processor activity was done, and is shown by a spreadsheet that follows the listing of each 960 code fragment. In these charts:

F represents a code fetch operation on the external bus,

F2 is the second fetch of a 2-word instruction on the external bus,

CF is a Cache Fetch,

D is a Decode operation,

EA is an Effective Address calculation,

A on B stands for the Address cycle for a data word on the external bus.

D on B stands for the Data cycle for a data word on the external bus

W is an extra clock (wait state) the author decided would be needed for a data write on the external bus,

X is any other kind of execution cycle, e.g., storing a value in a register

It's probable that these charts don't sufficiently account for limits on the number of instructions that can be pending in similar states simultaneously, and that as a result we made the 80960KA look slightly better than it should.

For the 80380, execution times were derived in two steps. First we simply added up the execution times listed in the User Manual, as for the 80186. Then the architect of the 380 analyzed the instruction flow, similarly to what was done for the 960, and added a few extra clocks for pipeline stalls and non-overlap between the Bus Interface Unit and Execution Unit. Because of this, perceptive readers may notice that the clocks shown for individual 80380 instructions don't always add up to the total execution figures.

DESCRIPTION OF THE CODE FRAGMENTS

I/O Loop

This code fragment reads received data, two bytes at a time, from a 16C30 Universal Serial Controller (USC), and stores the data in a memory buffer for each frame. The USC is the successor to Zilog's popular SCC, and has a 32-byte FIFO capacity. First, each sequence sets up whatever registers are needed to access the USC, the memory buffer, and a current pointer into the buffer named "rxi".

At the start of each loop, the code reads the number of bytes currently in the receive FIFO, from the MSbyte of a USC register called RICR. It also reads a 16-bit status register called RCSR.

IF there are no bytes left in the FIFO, the code exits from the fragment. If there is one byte in the RxFIFO, the code checks the status to see if the byte is either the last one of a frame, or is the byte at which a Receive Overrun condition occurred. If neither of these is the case, the code leaves the byte in the RxFIFO for the future, and exits from the fragment. Otherwise, or if there are two or more bytes in the FIFO, the code:

1. ensures that no interrupt can occur between the following steps,
2. reads two bytes from the FIFO via the USC register called RDR (the USC will only provide one if there's only one in the FIFO)
3. stores the data in memory at the address in the pointer "rxi",
4. increments rxi by 2,
5. stores rxi back in memory, and
6. enables interrupts to occur again.

After these operations the code tests the status obtained earlier from RCSR, and if the data just stored didn't represent the end of a frame, it goes back to the start of the loop described above. The following code calls an end-of-frame-handling subroutine called "_Handle_RxStatus"_ this part of the fragment counts toward the code memory required but not toward the execution time, because a frame ends only once in many executions of the loop.

Signed Byte Handling

This code fragment originally came from a customer code in the hard disk field. It examines three 8-bit variables in memory called NORM, Q, and K2. Actually NORM can range from -256 to +255 and is implemented as a 16-bit variable. It computes an eight-bit result in any of six ways, depending on the sign of NORM and how it compares to that of Q, as described in the comments at the top of each page of code.

First the code may access some or all or the three input variables and/or set up registers to point to one or more of them. Then it tests the sign of NORM, branching to the second "half" of the code fragment if it's positive. In each "half", the code compares NORM and Q and branches around in a tree-structured fashion to compute the result dictated by relative values of NORM and Q.

To evaluate the overall execution time of the fragment, we computed the execution time for each of the six result cases, and averaged them.

This may be the least clear code fragment as to its cosmic purpose, but it is a reasonable example of the kind of decision-tree processing that's typical of many I/O handling and control systems.

Multiply/Accumulate

This code fragments is also taken from a customer code in a hard-disk application. It uses four 16-bit input values in memory, CURSEC, POSN_ERR, S_GRAT, and K_GRAT, plus two memory tables of 16-bit values called S_TABLE and C_TABLE, each as large as the largest possible values of CURSEC. From these the code extracts S_TABLE (CURSEC) and saves the result in a memory variable S_VALUE, and similarly extracts C_TABLE (CUSEC) and saves it in K_VALUE. The code also multiplies each value by POSN_ERR, scales/divides each result by 64, and adds the results into memory variables S_ACCUM and K_ACCUM respectively, Finally it calculates R_CP= (S_VALUE*S_GRAT + K_VALUE*KGRAT) /32.

This code includes four 16x16 multiplications and 32-bit scale/shift operations. For all the processors except the CPU32, the fragment is coded to loop back once to minimize memory requirements, by taking advantage of the similarity of the computations for the "S" and "K" values.

Interrupt

These code fragments service a "receive status" interrupt from a Zilog 16C30 Universal Serial Controller (USC). The actual code size and execution time are reduced from a full-blown ISR, by evaluating for the case of a "Receive Overrun" event, and by isolating the details of handling an End of Frame event in a separate subroutine. This is done to emphasize the interrupt overhead for each processor, including interrupt latency, interrupt processing, context saving and restoring, and returning to the interrupted process.

Each code fragment saves register values and any other necessary contest info, then sets up a base address for the USC, clears the Interrupt Pending (IP) for Receive Status interrupts, and reads 16-bit status from the USC register RCSR. Then, if the overrun status bit is set, it writes two "command bytes" called "Enter Hunt Mode" and "Purge Rx" to USC registers. (These operations count are counted toward execution time.)

Next, if the status bit indicating the end of a frame is set, the code calls a subroutine to handle this condition. Neither the call nor the subroutine are counted toward execution time.

Next the code reads and writes several USC register to ready the device for future interrupts. Finally it restores the context and returns to the interrupted program.

The 80960KA does more operations automatically in hardware before and after the execution of the interrupt service routine proper, than do the other processors. The time to perform these operations were not specified in the Intel literature available to us, so the time was estimated in the first and last column of the execution chart, based on the time to do similar functional under software control.

Block Move Sequences

The "block move" sequences for all the processors are shown on one sheet. The 8096KA has no special instruction for this operation, but its Load and Store Quad Register instruction each handle 16 bytes per execution. The CPU32 has no special instruction either, but its two-word prefetch queue is capable of holding the two-instruction loop shown, so that no instruction fetches are needed for the duration of the block move, only data cycles. For 68020 we used the average of the Best Case and Cache Case, i.e., a cache hit with an "average" amount of instruction overlap.

Both the 80186 and 80380 have instructions intended for this purpose. The evaluation for 380 assumes that the global Longword (LW) control bit is set so that each iteration includes two 16-bit reads and two 16-bit writes.

SUMMARY

The final chart below summarizes and combines the memory requirements and execution times for each code fragment on the various processors clock speeds. The 80186 doesn't come in 25 or 40 MHz versions, so only 16 MHz results are shown. The CPU32, 68020, and 80960KA are shown for 16 and 25 MHz. The 80380 is shown for 16, 25, and 40 MHz clocking. In each case this includes the highest clock speed shown in the latest literature we could obtain for each processor family.

In all cases, the 80960KA has by far the largest code size, but makes up for it by needing the fewest clocks to execute. The 80186 has the smallest average code size, but makes up for that by being the slowest device for all cases except Block Move, for which it edged out the CPU32 to escape the cellar.

The CPU32 proved exemplary at the Multiply/Accumulate fragment, having the smallest code size and running second to the 80960KA for the faster execution time (even outperforming its 32-bit relative the 68020, due to its early-exit Booth multiplier). In the other cases it tended to run second-last to the 80960 in code size and to the 80186 in execution time.

The 68020 had the same code sizes as the CPU32 and improved on the CPU32's execution times, but perhaps not by enough to overcome the higher system costs of a 32-bit bus and memory subsystem

The Zilog 80380 typically ran close to the 80186 in code density and minimizing program size, as might be expected from an older architecture that was created when memory was more expensive than today. Perhaps more surprisingly, it finished second to the 80960KA in execution clocks most cases, and counting its faster clock rate ran competitively to the 960KA in total execution time.

When looking at a the normalized program size and execution time values in the summary table, remember that smaller values are better, and that a value less than 1 means that processor/clock rate combination is better than a 80380 at 25 MHz.

Of course there's something a little out of line about including the 80960KA in this comparison, which:

- costs far more than any of the other processors,
- entails added system-level expense because of its 32-bit data path and required memory width (also true of the 68020), and
- requires special "block transfer" memory design techniques

In fact, Intel has another member of its 80960 that is more like the other processors herein, the 8096KA. This device has a 16-bit data bus like the 80380 and 80186, and a more compact package that lowers its cost into a more competitive range. Unfortunately we were unable to obtain any timing information for the 80960SA in the time frame required for this benchmarking.

However, we did find an Intel brochure that allows the 80960SA to participate in these results in a small way. It showed a "Dhrystone" (fixed point) figure for the 80960SA of 12145, compared to 19740 for the 960KA. Multiplying the performance figures for the 960KA by 19740/12145 (smaller is better in our figures while larger is better for the Dhrystone) yielded the results shown in the third-last and last lines. For the last line that combines code size and execution time into a final figure of Merit, only the execution time values were scaled by Intel's Dhrystone results.

To wrap up, considering both code density and execution time for these code fragments, the new Zilog 80380 blows away other 16-bit processors including the 80960SA, and comes out about equal to the much more expensive 32-bit 80960KA if skewed by one speed grade (25 MHz 380 vs. 16 MHz 960, 40 MHz 380 vs. 25 MHz 960).

Note: *Due to the size of the coding of the bench marking exercise, only the CPU32 and the Z380 has been included in this paper. However, the methodology and presentation is similar for the other processors benchmarked. The complete benchmark report is available from Zilog in the Z380 CPU User's Manual.*

I/O LOOP: 68XXX–CPU32

```
        ; the following CPU32 code reads data from a USC
        ; this code is not warranted to be correct nor operative, and is
        ;  intended for performance benchmarking purposes only
        ; this version assumes that rxi variable is in first 64 Kbytes

Bytes Clks (CPU32)
—     —
4     8     MOVE.L     rxi,A1                ; address in rcv area
6     10    MOVE.L     #uscBase+ICR,A2       ; address of ICR in USC
      RxPoll16U_lp:
6     11    CMP.B  #1,RICR-ICR(A2)           ; <> 1 byte in RxFIFO?
4     7     MOVE.W     RCSR-ICR(A2),D0       ; get status
2     4/10  BHI    RxPoll16U_hav             ; around if > 1 byte
2     4/10  BLO    RxPoll16U_end             ; nothing to do if < 1 byte
4     5     AND.B  #$12,D0                   ; 1 byte: RxBound or overrun?
2     4/10  BZ     RxPoll16U_end             ; ignore 1 byte if neither
      RxPoll16U_hav:
4     9     BCLR   #7,(A2)                   ; disable interrupts
4     8     MOVE.W     RDR-ICR(A2),(A1)+     ; 2 serial bytes to Rx area
4     8     MOVE.L     A1,rxi                ; store rx pointer
4     9     BSET   #7,(A2)                   ; re-enable ints
4     5     AND.B  #$10,D0                   ; RxBound?
2     4/10  BZ     RxPoll16U_lp              ; loop if not
2           MOVEQ      #$C0,D0
4           AND.B  CCR+1-ICR(A2),D0          ; RSBs in use?
2           BNZ    RxPoll16U_rsb             ; around if so
4           MOVE.W     RCSR-ICR(A2),D0        ; take status from RCSR if not
2           BRA    RxPoll116U_call
      RxPoll16U_rsb:
4           MOVE.W     RDR-ICR(A2),D0         ; take status from RDR
      RxPoll16U_call:
4           BCLR   #1,D0
2           MOVE.W     D0,-(SP)
4           BCLR   #7,(A2)                   ; disable interrupts
4           JSR    _Handle_RxStatus          ; call the RxBound subroutine
2           ADDQ   #2,SP
4           BSET   #7,(A2)                   ; enable interrupts
2           BRA    RxPoll16U_lp              ; and loop
      RxPoll16U_end:

—     —
92    50+77*N clocks (CPU32)
```

I/O LOOP: CPU32

Bytes	Clks	Source	Hop	Top	Cop	LW	Hea1	Tea1	Cea1	Hea2	Tea2	Cea2
4	8	“MOVE.L rxi,A1”	0	2	4	2	1	3	5			
6	10	“MOVE.L #uscBase+ICR,A2”	2	4	8	2	1	3	5			
	18	subtotal: start										
6	11	“lp: CMP.B #1,RICR-ICR(A2)”	0	3	5	0	1	3	5	1	1	3
4	7	“MOVE.W RCSR-ISR(A2),D0”	0	0	2	0	1	3	5			
2	10	BHI hav (taken)	2	-2	8	0	0	0	0			
	4	BHI hav (not taken)	2	0	4	0	0	0	0			
2	10	BLO end (taken)	2	-2	8	0	0	0	0			
	4	BLO end (not taken)	2	0	4	0	0	0	0			
4	5	“AND.B #$12,D0”	0	0	2	0	1	1	3			
2	10	BZ end (taken)	2	-2	8	0	0	0	0			
	32	subtotal: exit										
	4	BZ end (not taken)	2	0	4	0	0	0	0			
4	9	“hav: BCLR #7,(A2)”	1	2	8	0	1	1	3			
4	8	“MOVE.W RDR-ICR(A2),(A1)+”	2	2	6	0	1	3	5			
4	8	“MOVE.L A1,rxi”	1	5	7	2	1	1	3			
4	9	“BSET #7,(A2)”	1	2	8	0	1	1	3			
4	5	“AND.B #$10,D0”	0	0	2	0	1	1	3			
2	10	BZ lp (taken)	2	-2	8	0	0	0	0			
	77	subtotal: loop										
	4	BZ lp (not taken)	2	0	4	0	0	0	0			
2		“MOVEQ #$C0,D0”										
4		“AND.B CCR+1-ICR(A2),D0”										
2		BNZ rsb										
4		“MOVE.W RCSR-ISR(A2),D0”										
2		BRA call										
4		“rsb: MOVE.W RDR-ICR(A2),D0”										
4		“BCLR #1,D0”										
2		“MOVE.W D0,-(SP)”										
4		“BCLR #7,(A2)”										
4		JSR _Handle_RxStatus										
2		“ADDQ #2,SP”										
4		“BSET #7,(A2)”										
2		BRA lp										
——	——	end:										
92	50+N*77		total									

I/O LOOP: 80380

```
        ; the following Z380 code reads data from a Zilog 16C30 USC.
        ; this code is not warranted to be correct nor operative, and is
        ;  intended for performance benchmarking purposes only
        ; this code assumes that the global LW and XM bits are set
        ;  and that the USC is in a 16-bit-addressed I/O space

Bytes Clks
—     —
 3     2      LD      DE,uscBase+RDR
 1     2      LD      B,D
 3     8      LD      HL,(rxi)                ; 32-bit address in variable
       RxPoll16U_lp:
 4     6*     INA     A,(uscBase+RICR+1)      ; get hi byte of RICR
 2     2*     SRL     A                       ; byte count to word count
 4     6*     INA     A,(uscBase+RCSR)        ; get status, no CC change
 2     2/6*   JR      NZ,RxPoll16U_hav
 2     2/6    JR      NC,RxPoll16U_end
 3     2      TST     12H                     ; RxBound or overrun?
 2     2/6    JR      Z,RxPoll16U_end
       RxPoll16U_hav:
 1     2*     DI
 2     7*     INIW                            ; 16 bits from RDR to buffer
 3     6*     LD      (rxi),HL                ; store address in buffer
 1     2*     EI
 3     2*     TST     10H
 2     6*     JR      Z,RxPoll16U_lp
 2            LD      C,CCR
 2            IN      A,(C)
 1                    LD      C,E             ; get status from RDR if RSBs
 3            TST     0C0H                    ; RSBs in use?
 2            JR      NZ,RxPoll16U_rsb        ; around if so
 2            LD      C,RCSR                  ; get status from RCSR if not
       RxPoll16U_rsb:
 2            INW     HL,(C)                  ; status word from RSB or RCSR
 2            RES     1,L                     ; leave overrun to int level
 1            DI
 2            PUSH    HL
 3            CALL    Handle_RxStatus
 1            INC     SP
 1            INC     SP
 1            EI
 2            JR      RxPoll16U_lp
RxPoll16U_end:

—     —
65    41+53N  (corrected for pipeline stalls)
```

SIGNED BYTE HANDLING: 68XXX–CPU32

```
        ; the following CPU32 code handles signed bytes.
        ; there are 3 signed byte variables in memory, Q, K2, and NORM.
        ;  Actually NORM can range from -256 to +255, so we test the
        ;  MSbyte of a 16-bit NORM but use only the LSbyte otherwise.
        ;  The result is as follows
        ; if      NORM < 0 then
        ;         if NORM > -Q then result := NORM
        ;         else if NORM > Q then result := -2*K2-NORM
        ;         else result := Q - K2
        ; else if NORM <= Q then result := NORM
        ;         else if NORM <= -Q then result := 2*K2-NORM
        ;         else result := K2 - Q
        ; Routines can leave the result wherever is most convenient.
        ; this code is not warranted to be correct nor operative, and
        ;  is intended for performance benchmarking purposes only.
        ; this code assumes that all variables are in the first 64K
        ;  bytes of memory

Bytes Clks (CPU32)
—     —
 4    7          MOVE.B   Q,D0              ; get variable
 4    7          MOVE.W   NORM,D1           ; get variable
 2    4/10       BPL.S    npos              ; around if positive
 2    2          NEG.B    D0                ; -Q
 2    2          CMP.B    D1,D0             ; -Q-NORM
 2    4/10       BMI.S    rnorm             ; go if -Q-NORM<0, NORM>-Q
 2    2          NEG.B    D0                ; Q
 2    2          CMP.B    D1,D0             ; Q-NORM
 2    4/10       BMI.S    m2k2              ; go if Q-NORM<0, NORM>Q
 4    7          SUB.B    K2,D0             ; Q - K2
 2    10         BRA.S    next
 4    7 m2k2:    MOVE.B   K2,D0             ; K2
 2    2          NEG.B    D0                ; -K2
 2    10         BRA.S    dmn
 2    2 rnorm:   MOVE.B   D1,D0             ; NORM
 2    10         BRA.S    next
 2    2  npos:   CMP.B    D1,D0             ; Q-NORM
 2    4/10       BPL      rnorm             ; go if Q-NORM>=0, NORM<=Q
 2    2          NEG.B    D0                ; -Q
 2    2          CMP.B    D1,D0             ; -Q-NORM
 2    4/10       BPL.S    p2k2              ; go if -Q-NORM>=0, NORM<=-Q
 4    7          ADD.B    K2,D0             ; K2 - Q
 2    10         BRA.S    next
 4    7 p2k2:    MOVE.B   K2,D0             ; K2
 2    2 dmn:     ADD.B    D0,D0             ; +- 2K2
 2    2          SUB.B    D1,D0             ; +- 2K2 - NORM
      next:
—     —
64    CPU32 68020
      48    NORM (pos)    40
      44    NORM (neg)    38
      55    2*K2-NORM     48
      63    -2*K2-NORM    56
      55    K2-Q          48
      51    Q-K2          46
      52.67 average       45.92
```

SIGNED BYTE HANDLING: CPU32

Bytes	Clks	Source	Hop	Top	Cop	Hea1	Tea1	Cea1	Hea2	Tea2	Cea2
4	7	“ move.b Q,D0”	0	0	2	1	3	5			
4	7	“ move.w NORM,D1”	0	0	2	1	3	5			
2	10	bpl.s npos (taken)	2	-2	8	0	0	0			
	4	bpl.s npos (not taken)	2	0	4	0	0	0			
2	2	neg.b D0	0	0	2	0	0	0			
2	2	“ cmp.b D1,D0”	0	0	2	0	0	0			
2	10	bmi.s rnorm (taken)	2	-2	8	0	0	0			
	4	bmi.s rnorm (not taken)	2	0	4	0	0	0			
2	2	neg.b D0	0	0	2	0	0	0			
2	2	“ cmp.b D1,D0”	0	0	2	0	0	0			
2	10	bmi.s m2k2 (taken)	2	-2	8	0	0	0			
	4	bmi.s m2k2 (not taken)	2	0	4	0	0	0			
4	7	“ sub.b k2,d0”	0	0	2	1	3	5			
2	10	bra.s next	2	-2	8	0	0	0			
4	7	“m2k2: move.b K2,d0”	0	0	2	1	3	5			
2	2	neg.b D0	0	0	2	0	0	0			
2	10	bra.s dmn	2	-2	8	0	0	0			
2	2	“rnorm: move.b D1,D0”	0	0	2	0	0	0			
2	10	bra.s next	2	-2	8	0	0	0			
2	2	“npos: cmp.b D1,D0”	0	0	2	0	0	0			
2	10	bpl rnorm (taken)	2	-2	8	0	0	0			
	4	bpl rnorm (not taken)	2	0	4	0	0	0			
2	2	neg.b D0	0	0	2	0	0	0			
2	2	“ cmp.b D1,D0”	0	0	2	0	0	0			
2	10	bpl.s p2k2 (taken)	2	-2	8	0	0	0			
	4	bpl.s p2k2 (not taken)	2	0	4	0	0	0			
4	7	“ add.b K2,D0”	0	0	2	1	3	5			
2	10	bra.s next	2	-2	8	0	0	0			
4	7	“p2k2: move.b k2,d0”	0	0	2	1	3	5			
2	2	“dmn: add.b d0,d0”	0	0	2	0	0	0			
2	2	“sub.b d1,d0”	0	0	2	0	0	0			
——	——	next:									
64	48	NORM (pos)									
	44	NORM (neg)									
	55	2*K2-NORM									
	63	-2*K2-NORM									
	55	K2-Q									
	51	Q-K2									
	52.67	average									

SIGNED BYTE HANDLING: 80380

```
        ; the following Z380 code handles signed bytes.
        ; there are 3 signed byte variables in memory, Q, K2, and NORM.
        ;  Actually NORM can range from -256 to +255, so we test the
        ;  MSbyte of a 16-bit NORM but use only the LSbyte otherwise.
        ;  The result is as follows
        ; if      NORM < 0 then
        ;         if NORM > -Q then result := NORM
        ;         else if NORM > Q then result := -2*K2-NORM
        ;         else result := Q - K2
        ; else if NORM <= Q then result := NORM
        ;         else if NORM <= -Q then result := 2*K2-NORM
        ;         else result := K2 - Q
        ; Routines can leave the result wherever is most convenient.
        ; this code is not warranted to be correct nor operative, and is
        ;  intended for performance benchmarking purposes only
        ; this code assumes the global LW bit is cleared

Bytes Clks
 3     6          LD      A,(Q)           ; get variable
 4     2          LD      HL,K2           ; address of variable
 3     6          LD      BC,(NORM)       ; get variable
 1     2          OR      B,B             ; test if NORM positive
 2     2/6        JR      Z,npos          ; around if so
 2     2          NEG     A               ; -Q
 1     2          CP      A,C             ; -Q-NORM
 3     2/6        JP      S,rnorm         ; go if -Q-NORM<0, NORM>-Q
 2     2          NEG     A               ; Q
 1     2          CP      A,C             ; Q-NORM
 3     2/6        JP      S,m2k2          ; go if Q-NORM<0, NORM>Q
 1     6          SUB     A,(HL)          ; Q - K2
 2     6          JR      next
 1     6 m2k2:    LD      A,(HL)          ; K2
 2     2          NEG     A               ; -K2
 2     6          JR      dmn
 1     2 rnorm:   LD      A,C             ; NORM
 2     6          JR      next
 1     2 npos:    CP      A,C             ; Q-NORM
 3     2/6        JP      NS,rnorm        ; go if Q-NORM>=0, NORM<=Q
 2     2          NEG     A               ; -Q
 1     2          CP      A,C             ; -Q-NORM
 3     2/6        JP      NS,p2k2         ; go if -Q-NORM>=0, NORM<=-Q
 1     6          ADD     A,(HL)          ; K2 - Q
 2     6          JR      next
 1     6 p2k2:    LD      A,(HL)          ; K2
 1     2 dmn:     ADD     A,A             ; +- 2K2
 1     2          SUB     A,C             ; +- 2K2 - NORM
       next:
—      —
52      NORM (pos)      36
        NORM (neg)      38
        2*K2-NORM       46
        -2*K2-NORM      50
        K2-Q            44
        Q-K2            42

        average         42.67
```

MULTIPLY/ACCUMULATE: 68XXX–CPU32

```
        ; this CPU32 code performs a 16-bit multiply/accumulate:
        ; several 16-bit variables pre-exist in memory, including
        ;  CURSEC, POSN_ERR, S_GRAT, and K_GRAT.  In addition,
        ;  two tables S_TABLE and C_TABLE are of a size equal to
        ;  the possible range of values of CURSEC.  16-bit results
        ;  of this calculation in memory include S_VALUE, K_VALUE,
        ;  R_CP, and two accumulators S_ACCUM and K_ACCUM:

        ;  S_VALUE := S_TABLE(CURSEC)
        ;  K_VALUE := C_TABLE(CURSEC)
        ;  S_ACCUM := S_ACCUM + ((S_VALUE*POSN_ERR)/64)
        ;  K_ACCUM := K_ACCUM + ((K_VALUE*POSN_ERR)/64)
        ;  R_CP := (S_VALUE*S_GRAT + K_VALUE*K_GRAT) / 32

        ; to optimize memory accessing, all routines may assume
        ;  that variables S_VALUE, S_GRAT, S_ACCUM, K_VALUE, K_GRAT,
        ;  K_ACCUM are consecutive in memory in whatever order is
        ;  optimal for their instruction set, while CURSEC. POSN_ERR,
        ;  S_TABLE, and C_TABLE are at unrelated locations.  R_CP
        ;  can be in either place.

        ; the order in this version is S_VALUE, S_ACCUM, S_GRAT,
        ;  K_VALUE, K_ACCUM, K_GRAT, R_CP.

        ; this code is not warranted to be correct nor operative, and is
        ;  intended for performance benchmarking purposes only

        ; the size/clocks figures assume all data is in the first
        ;  64K bytes

Bytes Clks (CPU32)
————  —
 4     7     MOVE.W      CURSEC,D0
 6     10    MOVE.W      S_TABLE(D0.W*2),D1      ; get S_VALUE from table
 4     5     LEA         S_VALUE,A0              ; start pointer into variables
 2     5     MOVE.W      D1,(A0)+                ; store S_VALUE
 2     2     MOVE.W      D1,D2                   ; copy it
 4     31    MULS.W      POSN_ERR,D1
 2     6     ASR.L       #6,D1                   ; divide by 64
 2     7     ADD.W       D1,(A0)+                ; add into accumulator
 2     29    MULS.W      (A0)+,D2                ; S_GRAT*S_VALUE
 6     10    MOVE.W      C_TABLE(D0.W*2),D1      ; get K_VALUE from table
 2     5     MOVE.W      D1,(A0)+                ; store K_VALUE
 2     2     MOVE.W      D1,D0                   ; copy it
 4     31    MULS.W      POSN_ERR,D1
 2     6     ASR.L       #6,D1                   ; divide by 64
 2     7     ADD.W       D1,(A0)+                ; add into accumulator
 2     29    MULS.W      (A0)+,D0                ; K_GRAT*K_VALUE
 2     2     ADD.L       D2,D0                   ; S_GRAT*S_VALUE + K_GRAT*K_VALUE
 2     6     ASR.L       #5,D0                   ; /32
 2     4     MOVE.W      D0,(A0)+                ; save that in R_CP
—      —
54     204 clocks (CPU32)
       212 clocks (68020)
```

MULTIPLY/ACCUMULATE: CPU32

Bytes	Clks	Source	Hop	Top	Cop	Hea1	Tea1	Cea1
4	7	“MOVE.W CURSEC,D0”	0	0	2	1	3	5
6	10	“MOVE.W S_TABLE(D0.W*2),D1”	0	0	2	2	2	8
4	5	“LEA S_VALUE,A0”	0	0	2	1	1	3
2	5	“MOVE.W D1,(A0)+”	1	1	5	0	0	0
2	2	“MOVE.W D1,D2”	0	0	2	0	0	0
4	31	“MULS.W POSN_ERR,D1”	0	0	26	1	3	5
2	6	“ASR.L #6,D1”	4	0	6	0	0	0
2	7	“ADD.W D1,(A0)+”	0	3	5	1	1	3
2	29	“MULS.W (A0)+,D2”	0	0	26	1	1	3
6	10	“MOVE.W C_TABLE(D0.W*2),D1”	0	0	2	2	2	8
2	5	“MOVE.W D1,(A0)+”	1	1	5	0	0	0
2	2	“MOVE.L D1,D0”	0	0	2	0	0	0
4	31	“MULS.W POSN_ERR,D1”	0	0	26	1	3	5
2	6	“ASR.L #6,D1”	4	0	6	0	0	0
2	7	“ADD.W D1,(A0)+”	0	3	5	1	1	3
2	29	“MULS.W (A0)+,D0”	0	0	26	1	1	3
2	2	“ADD.L D2,D0”	0	0	2	0	0	0
2	6	“ASR.L #5,D0”	4	0	6	0	0	0
2	4	“MOVE.W D0,(A0)+”	1	1	5	0	0	0
54	204							

MULTIPLY/ACCUMULATE: 80380

```
        ; this 80380 code performs a 16-bit multiply/accumulate:
        ; several 16-bit variables pre-exist in memory, including
        ;  CURSEC, POSN_ERR, S_GRAT, and K_GRAT.  In addition,
        ;  two tables S_TABLE and C_TABLE are of a size equal to
        ;  the possible range of values of CURSEC.  16-bit results
        ;  of this calculation in memory include S_VALUE, K_VALUE,
        ;  R_CP, and two accumulators S_ACCUM and K_ACCUM:

        ;  S_VALUE := S_TABLE(CURSEC)
        ;  K_VALUE := C_TABLE(CURSEC)
        ;  S_ACCUM := S_ACCUM + ((S_VALUE*POSN_ERR)/64)
        ;  K_ACCUM := K_ACCUM + ((K_VALUE*POSN_ERR)/64)
        ;  R_CP := (S_VALUE*S_GRAT + K_VALUE*K_GRAT) / 32

        ; to optimize memory accessing, all routines may assume
        ;  that variables S_VALUE, S_GRAT, S_ACCUM, K_VALUE, K_GRAT,
        ;  K_ACCUM are consecutive in memory in whatever order is
        ;  optimal for their instruction set, while CURSEC. POSN_ERR,
        ;  S_TABLE, and C_TABLE are at unrelated locations.  R_CP
        ;  can be in either place.

        ; the order in this version in S_VALUE, S_ACCUM, S_GRAT, K_VALUE,
        ;  K_ACCUM, K_GRAT, R_CP.

        ; this code is not warranted to be correct nor operative, and is
        ;  intended for performance benchmarking purposes only
        ; this code assumes that the global LW and XM bits are cleared.

Bytes Clks
————  —
 4      6        LD        IX,(CURSEC)
 2      2        ADD       IX,IX
 2      2        DDIR      IB
 5      8        LD        HL,(IX+S_TABLE)          ; get S_VALUE from table
 2      2        DDIR      IB
 5      8        LD        IY,(IX+C_TABLE)          ; get K_VALUE from table
 3      2=35     LD        DE,S_VALUE               ; start pointer into variables
2       3 lp:    LD        (DE),HL                  ; save VALUE in memory
 2      2        LD        IX,HL                    ; save in reg
 4      6        LD        BC,(POSN_ERR)
 3      10       MULTW     HL,BC                    ; VALUE * POSN_ERR (16x16=32)
 2      2        DDIR      LW
 1      2        ADD       HL,HL
 2      2        DDIR      LW
 1      2        ADD       HL,HL
 1      2        LD        A,H
 2      2        SWAP      HL
 1      2        LD        H,L
 1      2        LD        L,A                      ; 16 bit product/64
 1      2        INC       DE
 1      2        INC       DE
 2      6        LD        BC,(DE)                  ; get accum
 1      2        ADD       HL,BC                    ; add
 2      3        LD        (DE),HL                  ; save accum
```

MULTIPLY/ACCUMULATE: 80380 (Continued)

```
Bytes   Clks
 1      2          INC       DE
 1      2          INC       DE
 2      6          LD        HL,(DE)                ; get GRAT
 1      2          INC       DE
 1      2          INC       DE
 2      2          LD        BC,IX                  ; retrieve value
 3      10         MULTW     BC                     ; GRAT*VALUE
 2      2          LD        A,K_VALUE MOD 256
 1      2          CP        A,E
 2      2/6=89     JR        NZ,kdone               ; around if K group done

 2      2          DDIR      LW
 2      3          EX        HL,IY                  ; HL:=K_VALUE, IY:=S_VALUE*S_GRAT

 2      6=11       JR        lp                     ; and go do K group

 2      2 kdone:   DDIR      LW
 2      2          ADD       HL,IY                  ; S_VALUE*S_GRAT + K_VALUE*K_GRAT
 2      2          DDIR      LW
 1      2          ADD       HL,HL
 2      2          DDIR      LW
 1      2          ADD       HL,HL
 2      2          DDIR      LW
 1      2          ADD       HL,HL                  ; 32-bit left shift 3
 1      2          LD        A,H
 2      2          SWAP      HL
 1      2          LD        H,L
 1      2          LD        L,A                    ; sum div 32
 3      6=30       LD        (R_CP),HL              ; save that
—       —
95      254 (35+89+11+89+30)
```

INTERRUPT: 68XXX–CPU32

```
        ; This CPU32 code handles Rx Status interrupts from a 16C30.
        ; It is evaluated for an overrun condition, so that End Of
        ;  Frame processing, which is handled by a separate subroutine,
        ;  doesn't count toward the totals.
        ; It is not warranted to be correct nor operative, and is
        ;  intended for performance benchmarking purposes only

        ; It assumes the USC is in a 24-bit addressed memory space
        ;  and that the hardware includes byte/word addressing
        ;  hardware (i.e., an environment like the IUSC/AT Starter Kit)

Bytes Clks (CPU32)
—     —
      32  interrupt (per CPU32 ref man p.8-27)
      rxStInt:
       ; save registers
4     73    MOVEM.L A0-6/D0-7,-(SP)      ; could save less, but we don't
       ; begin handling the interrupt    know what procEOF does...
6     7     LEA     uscBase,A0
6     10    MOVE.B  #clrIP+RS_IP,DCCR(A0)      ; clear IP
4     7     MOVE.W  RCSR(A0),D0 ; get status
4     4     BTST    #rxOv,D0          ; test overflow
2     4     BEQ     noOver            ; around if not
       ; handle Rx overrun
6     10    MOVE.B  #EnterHuntMode,RCSR+1(A0) ; force Rx into Hunt
6     12    OR.B    #PurgeRx,CCAR+1(A0) ; issue purge Rx command
       ; handle RxBound (end of frame)
4     4     BTST    #rxBnd,D0
2     10    BZ      noEOF             ; around if no End of Frame
4           BSR     procEOF                   ; call subr if so
       ; clear interrupt hardware
4     5 noEOF: AND.B        #$F6,D0                   ; mask status
4     6     MOVE.B  D0,RCSR(A0) ; unlatch status bits we saw
4     7     MOVE.B  RICR(A0),D0  ; save arm bits
4     6     CLR.B   RICR(A0)          ; disarm all
4     6     MOVE.B  D0,RICR(A0)  ; rearm
6     10    MOVE.B  #clrIUS+RS_IUS,DCCR+1(A0)
       ; restore regs, dismiss interrupt and return
4     74    MOVEM.L (SP)+,A0-6/D0-7
2     26    RTE
—     —
80    313 clocks (CPU32)
      288 clocks (68020)
```

INTERRUPT: 80380

```
        ; This 380 code handles Rx Status interrupts from a 16C30.
        ; It is evaluated for an overrun condition, so that End Of
        ;  Frame processing, which is handled by a separate subroutine,
        ;  doesn't count toward the totals.
        ; It is not warranted to be correct nor operative, and is
        ;  intended for performance benchmarking purposes only

        ; It assumes the USC is in a 24-bit addressed memory space
        ;  and that the hardware includes byte/word addressing
        ;  hardware (i.e., an environment like the IUSC/AT Starter Kit)

Bytes Clks
—— ——
        18 (interrupt time)
        rxStInt:
         ; save registers
 2      2       DDIR    LW
 2      6       PUSH    SR              ; save old control settings
 3      4       LDCTL   SR,intBank      ; one reg bank dedicated
                                        ; for unnested interrupts
         ; begin handling the interrupt
 2      2       DDIR    IB
 5      4       LD      IX,uscBase      ; set 24-bit address of USC
 4      6       LD      (IX+DCCR),clrIP+RS_IP ; clear IP bit
 4      7       LD      BC,(IX+RCSR)  ; get status
 2      2       BIT     rxOv,C
 2      2/6     JR      Z,noOver        ; around if no overflow flag
         ; handle Rx overrun
 4      6       LD      (IX+RCSR+1),EnterHuntMode ; force Rx hunt mode
 3      7       LD      A,(IX+CCAR+1)
 2      2       OR      A,PurgeRx
 3      6       LD      (IX+CCAR+1),A; issue purge Rx command
         ; handle RxBound (End of Frame)
 2      2/6 noOver:BIT  rxBd,C
 3      2       CALL    NZ,procEOF      ; call End of Frame procedure
         ; clear interrupt hardware
 2      2       AND     C,0F6H
 3      6       LD      (IX+RCSR),C     ; unlatch status bits we saw
 3      7       LD      A,(IX+RICR)     ; get IA bits
 4      6       LD      (IX+RICR),0     ; drop IA bits
 3      6       LD      (IX+RICR),A     ; rearm them
 4      6       LD      (IX+DCCR+1),clrIUS+RS_IUS    ; clear IUS
         ; restore registers, dismiss interrupt and return
 2      2       DDIR    LW
 2      8       POP     SR
 2      8       RETI
——      ——
66      133
```

Summary of Benchmarks

Normalized to 25MHz 80380										
Proc	i 80186	CPU32	CPU32	68020	68020	Z380	Z380	Z380	80960KA	80960KA
clock rate, MHz	16	16	25	16	25	16	25	40	16	25
clk period, nS	62.5	62.5	40	62.5	40	62.5	40	25	62.5	40
I/O Loop (bytes)	61	92	92	92	92	65	65	65	120	120
Bytes, Z380=1	0.94	1.42	1.42	1.42	1.42	1.00	1.00	1.00	1.85	1.85
I/O Loop (formula)	60+80*N	50+77*N	50+77*N	56+48N	56+48N	41+53*N	41+53*N	41+53*N	56+24*N	56+24*N
I/O Loop (clks @ N=1)	140	127	127	104	104	94	94	94	80	80
I/O Loop (nS @ N=1)	8750	7938	5080	6500	4160	5875	3760	2350	5000	3200
nS, N=1, 25MHz Z380=1	2.33	2.11	1.35	1.73	1.11	1.56	1.00	0.63	1.33	0.85
I/O Loop (clks @ N=8)	700	666	666	440	440	465	465	465	248	248
I/O Loop (nS @ N=8)	43750	41625	26640	27500	17600	29063	18600	11625	15500	9920
nS, N=8, 25MHz Z380=1	2.35	2.24	1.43	1.48	0.95	1.56	1.00	0.63	0.83	0.53
signed bytes (bytes)	63	64	64	64	64	52	52	52	76	76
bytes, Z380=1	1.21	1.23	1.23	1.23	1.23	1.00	1.00	1.00	1.46	1.46
signed bytes (clks)	79	53	53	46	46	43	43	43	31	31
signed bytes (nS)	4917	3292	2107	2875	1840	2667	1707	1067	1917	1227
nS, 25MHz Z380=1	2.88	1.93	1.23	1.68	1.08	1.56	1.00	0.63	1.12	0.72
multiply/accum (bytes)	72	54	54	54	54	95	95	95	104	104
bytes (Z380=1)	0.76	0.57	0.57	0.57	0.57	1.00	1.00	1.00	1.09	1.09
multiply/accum (clks)	404	204	204	212	212	254	254	254	92	92
multiply/accum (nS)	25250	12750	8160	13250	8480	15875	10160	6350	5750	3680
nS, 25MHz Z380=1	2.49	1.25	0.80	1.30	0.83	1.56	1.00	0.63	0.57	0.36
interrupt (bytes)	63	80	80	80	80	66	66	66	92	92
bytes (Z380=1)	0.95	1.21	1.21	1.21	1.21	1.00	1.00	1.00	1.39	1.39
interrupt (clks)	328	313	313	288	288	133	133	133	123	123
interrupt (nS)	20500	19563	12520	18000	11520	8313	5320	3325	7688	4920
nS, 25MHz Z380=1	3.85	3.68	2.35	3.38	2.17	1.56	1.00	0.63	1.45	0.92
Block move, clks/byte	4.00	4.25	4.25	2.875	2.875	2.75	2.75	2.75	1.25	1.25
Block move, nS/byte	250	266	170	180	115	172	110	69	78	50
nS, 25MHz Z380=1	2.27	2.41	1.55	1.63	1.05	1.56	1.00	0.63	0.71	0.45
Bytes, ave of Z380=1	0.97	1.11	1.11	1.11	1.11	1.00	1.00	1.00	1.45	1.45
nS, ave of 25 MHz Z380=1	2.70	2.27	1.45	1.87	1.20	1.56	1.00	0.63	1.00	0.64
est for 80960SA*									1.63	1.04
ave of all 25MHz Z380=1	2.00	1.81	1.31	1.56	1.16	1.34	1.00	0.78	1.18	0.96
est for 80960SA*									1.56	1.20
* 80960SA times estimated per intel's Dhrystone figures: 19740 for KA, 12145 for SA										

Warning: DO NOT USE IN LIFE SUPPORT

LIFE SUPPORT POLICY

ZILOG'S PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS PRIOR WRITTEN APPROVAL OF THE PRESIDENT AND GENERAL COUNSEL OF ZILOG CORPORATION.

As used herein

Life support devices or systems are devices which (a) are intended for surgical implant into the body, or (b) support or sustain life and whose failure to perform when properly used in accordance with instructions for use provided in the labeling can be reasonably expected to result in a significant injury to the user. A critical component is any component in a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system or to affect its safety or effectiveness.

Document Disclaimer